Concurrent programming

Introduction

Bibliography

- *M. Ben-Ari - "Principles of Concurrent and Distributed Programming"*
- *W. Richard Stevens – "UNIX Network Programming vol.1 and 2"*
- *A.S. Tanenbaum - "Distributed Operating Systems"*
- Z. Manna, A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems (Specific ation)*, Springer-Verlag, 1992
- Andrew Troelsen - various books

Bibliography

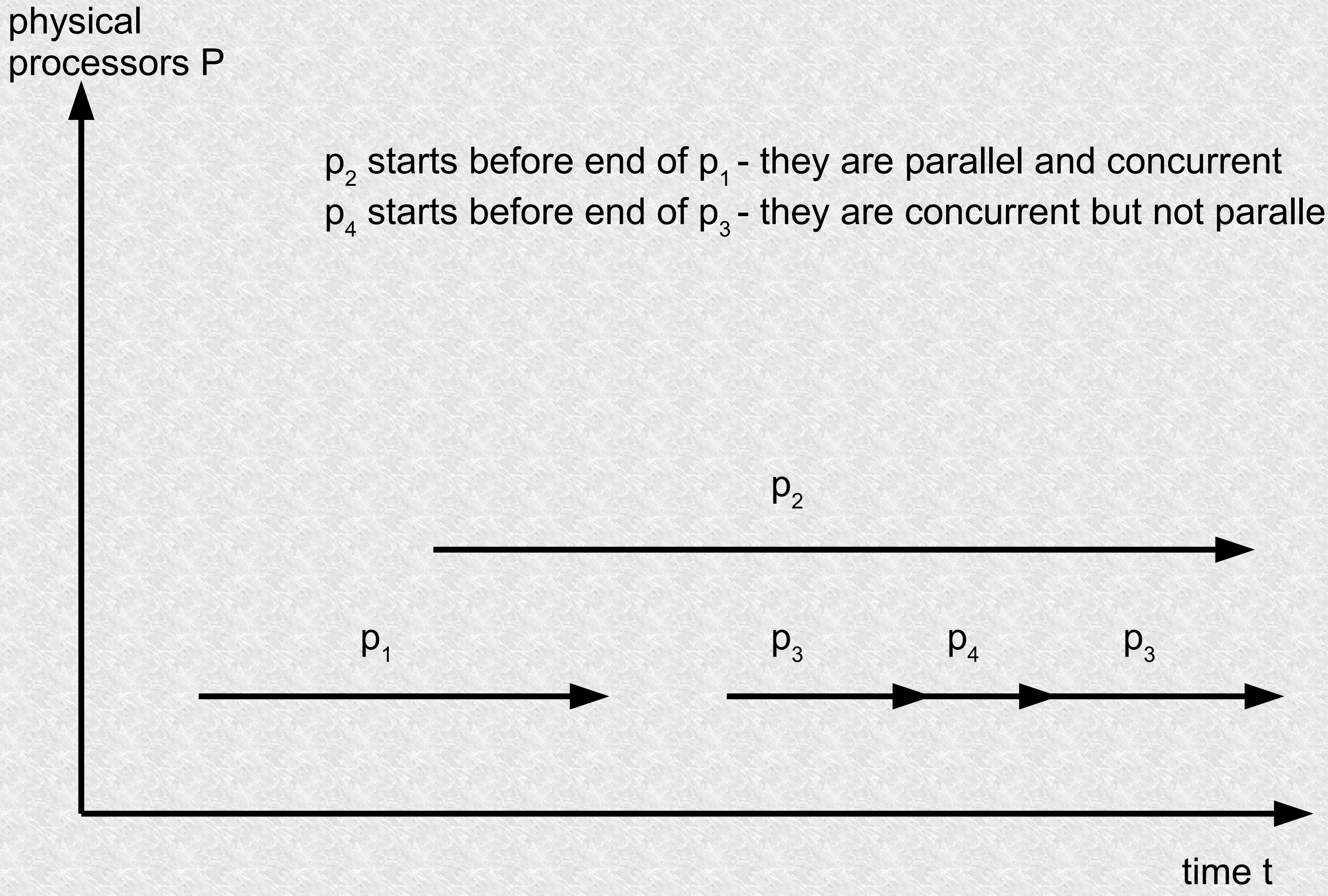
- **http://www.albahari.com/threading/**
- https://docs.microsoft.com/plpl/dotnet/standard/asynchronous-programmingpatterns/?view=netframework-4.7.2
- https://docs.microsoft.com/pl-pl/dotnet/standard/asynchronous-programming-patterns/task-based-asynchronous-pattern-tap?view=netframework-4.7.2

Basic definitions

- **Process** – a sequence program in progress
- **Concurrent processes –** processes that may (but need not) run in parallel. One process must begin before the end of the other one
- **Parallel processes** – concurrent processes executing at the same time

Parallelism and concurrency

p_2 starts before end of p_1 - they are parallel and concurrent
p_4 starts before end of p_3 - they are concurrent but not parallel



Basic definitions

- **Concurrent program** – a program consisting of several sequential processes that usually transfer some data to each other or just synchronize
- **Concurrent programming -** creating programs, the execution of which causes a certain number of concurrent processes (usually these processes are dependent)

Basic definitions

- **Events**
  - **Synchronous –** the ones we are waiting for
  - **Asynchronous –** occur unexpectedly at any time
- **The atomic instruction** - which cannot be broken, indivisible

Basic definitions

- **Dependent processes –** two processes are called dependent if the execution of either of them affects the execution of the other
- **Shared variable** – common variable, used by several concurrent processes
- **Critical section** – for example, part of the process where it uses a shared variable or common resource
- **Synchronization** – arranging the actions of individual processes in time

Basic definitions

In the literature, you can also meet the term - **Distributed** - when computing is realizing on many remote computers

Correctness of Concurrent Programs

Concurrent program is **correct** when has **safety** properties (properties that must always hold) and **liveness** properties(that must eventually hold)

• **safety properties - ensuring** that there is no collision and all data is correct. [example Unsafeinteriving]

• **liveness properties** – if any of the processes is waiting for an event, it will take place in a finite time. A special kind of liveness property is called the fairness property.

Basic definitions

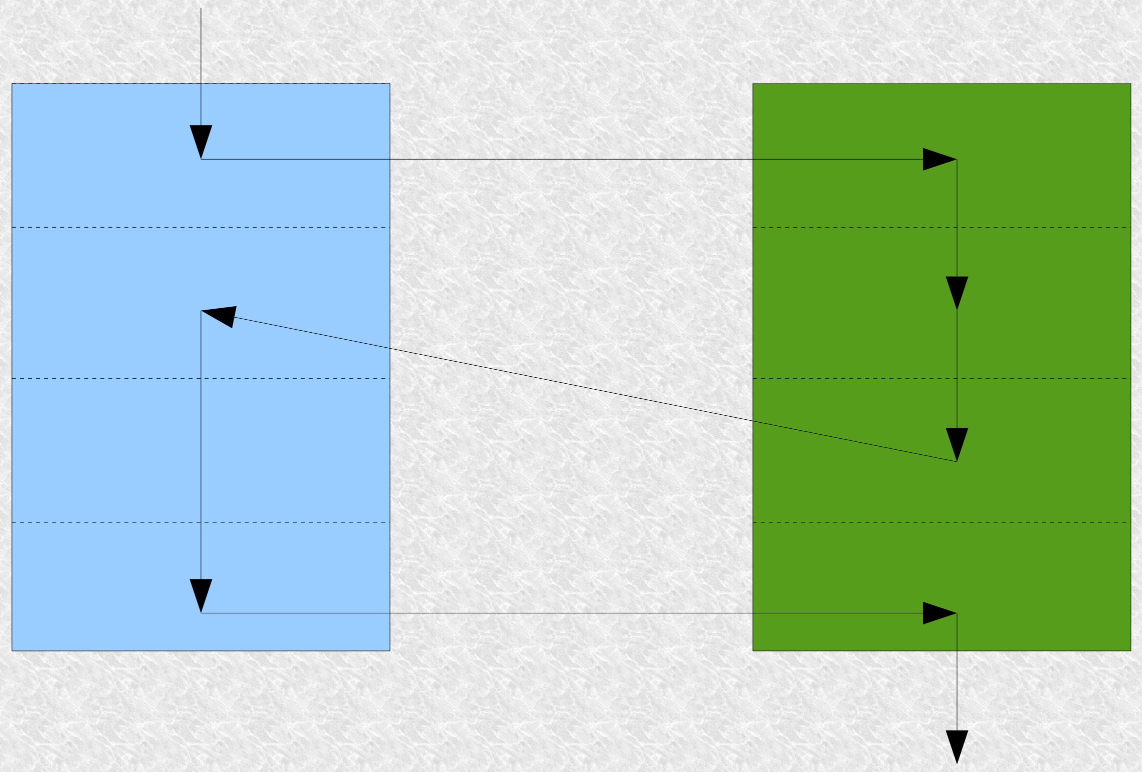
- **Weak Fairness** - If a thread continually makes a request (one time) it will eventually be granted.
- **Strong Fairness -** If a thread makes a request infinitely often (many times) it will eventually be granted.
- **Linear Waiting** - If a thread makes a request, it will be granted before any other thread is granted a request more than once.
- **FIFO** – If a thread makes a request it will be granted before any other thread making a later request.

Common mistakes

- **Deadlock –** Two or more processes from the collection *P* are waiting for an event that only the other blocked thread from collection *P* can generate.
- **Livelock** - when two processes try to get to the critical section at the same time and they give up for an equal moment and try again and again.
- **Starvation** – A situation in which a process is infinitely suspended because the event it is waiting for causes other processes to resume
- **Active waiting** - the process waiting for the event constantly checks if it has already occurred, unnecessarily using the CPU time.
  [example ActiveWaiting]

Basic definitions

- **Interleaving** - The abstraction of concurrent programming consists in examining the interleaving sequences of the execution of atomic instructions of sequential processes



Basic definitions

- For 2 processes consiting respectivly N and M atomic instructions we have

$$\frac{(N+M)!}{N!\,M!}$$

- For previous example we have "only" **70** combinations.
- For example 2 processes with 10 atomic instruction for each, we have **184 756** combinations.
- But when we have 15 atomic instructions **1 307 674 368 000**

Simple increment instruction

Some interleavings are bad

- LOAD n;
- ADD 1;
- STORE n;

- LOAD n;
- ADD 1;
- STORE n;

LOAD n; LOAD n; ADD 1; ADD 1; STORE n; STORE n;

LOAD n; ADD 1; LOAD n; ADD 1; STORE n; STORE n;

Basic definitions

We consider only two cases

- **Competition** - Two processes are competing for the same resource: computing resource, memory cell, or communication channel
- **Communication** - Two processes may want to communicate to transfer data from one to the other

All local sections are treated as one atomic instruction

Time dependencies

Remember!!!
Processes can run at any speed and can respond to any external signal! No time dependencies!
You can't expect that one process ends before other one only that it counts faster.

Classic Problems

- Mutual exclusion
- Producer and consumer
- Readers and writers
- Five philosophers
- Byzantine generals

Distributed programming models

Types of communication

- **Synchronous communication** - necessary sender and recipient for the exchange of messages
- **Asynchronous communication** - after sending the message, the sender does not have to wait for the recipient to receive it.

Identifying processes and data flow

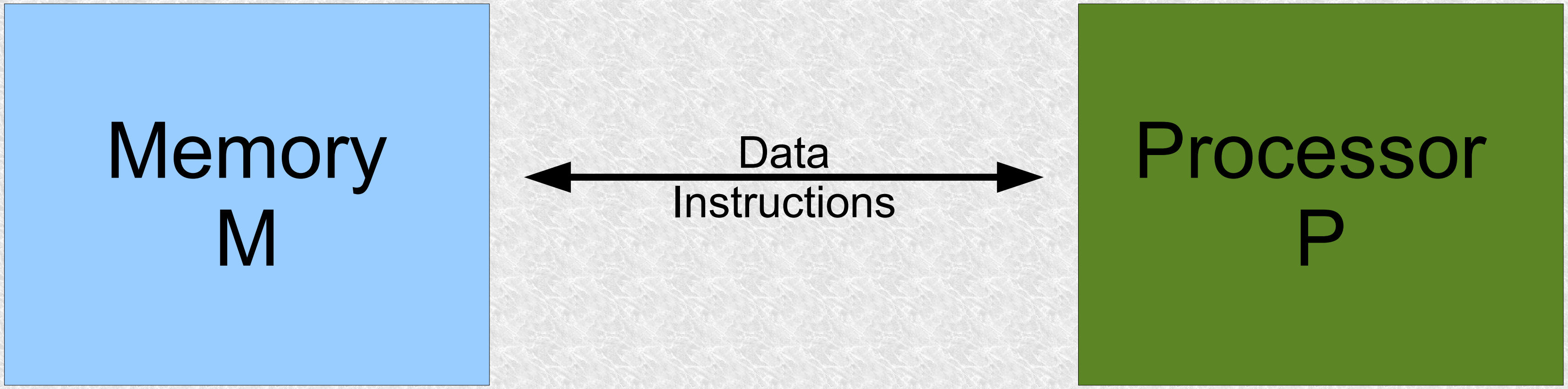
- **Dedicated channels** - both the sender and recipient know their identifiers. Each message is transferred without any additional costs related to e.g. address recalculation.
- **Asymmetric communication** - The sender knows the receiver's address, but the receiver does not need to know it. It is very well suited for client-server systems
- **Broadcast messages** - Both the recipient does not know from whom to receive the message and the sender does not know the recipient, so it sends it to everyone.

Creating of processes

- Dynamic
  - Flexibility
  - Dynamic resource use
  - Load balancing
- Static
  - Quick initiation
  - Specialized tasks

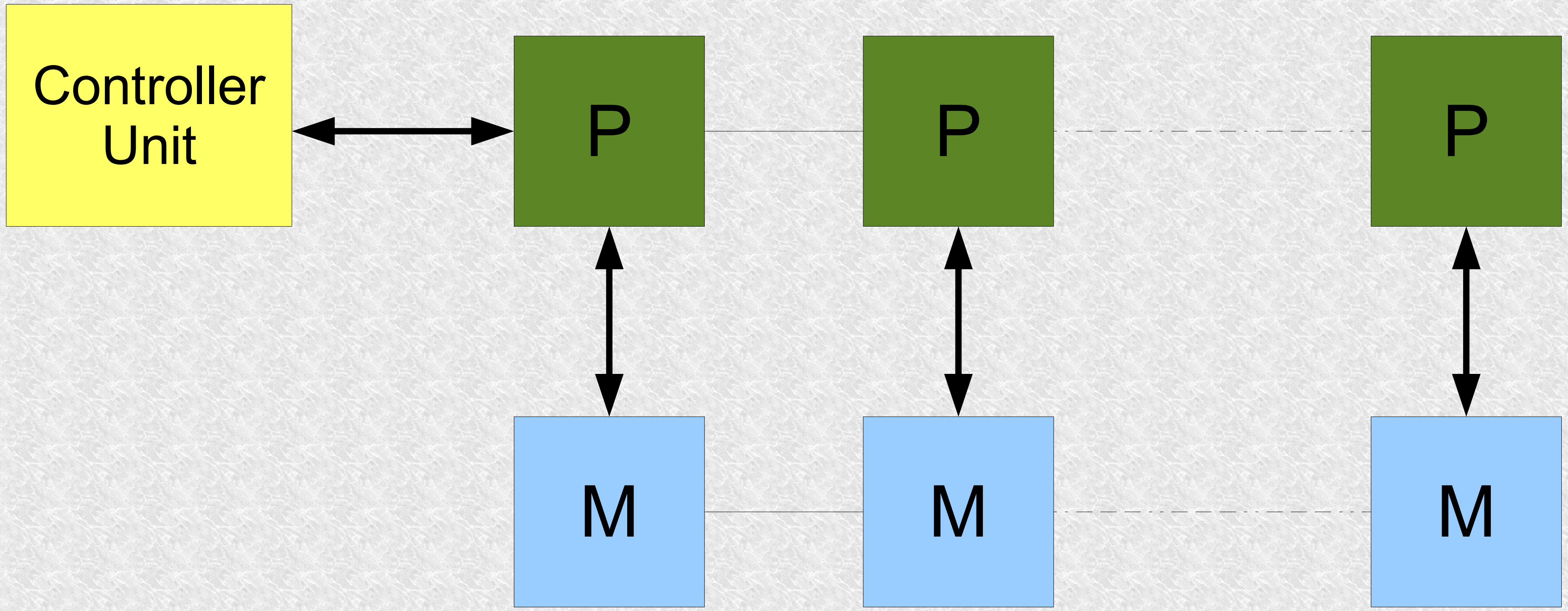
Classification of parallel machines

SISD – Single Instruction Stream, Single Data Stream



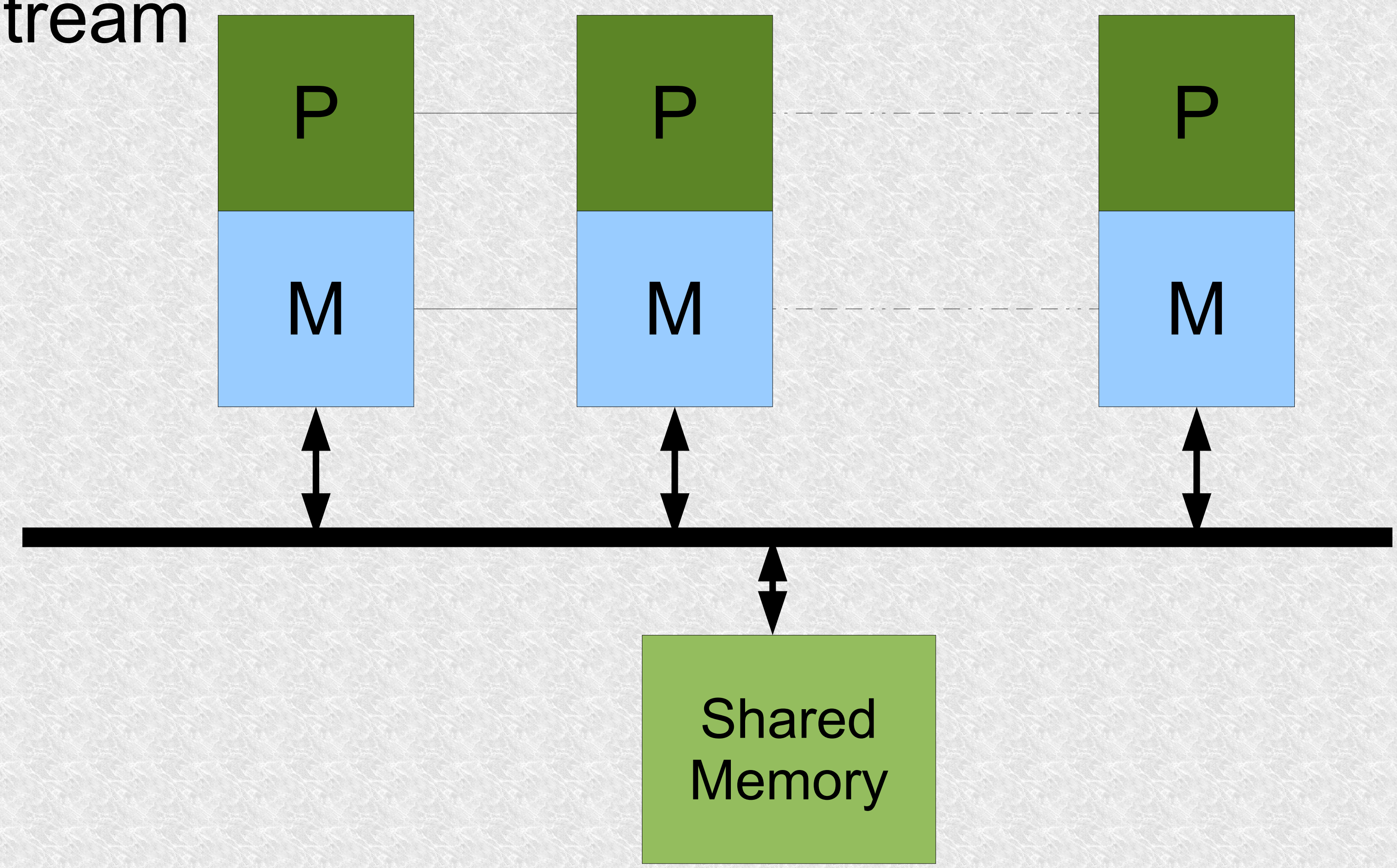
Classification of parallel machines

SIMD – Single Instruction Stream, Multiple Data Stream



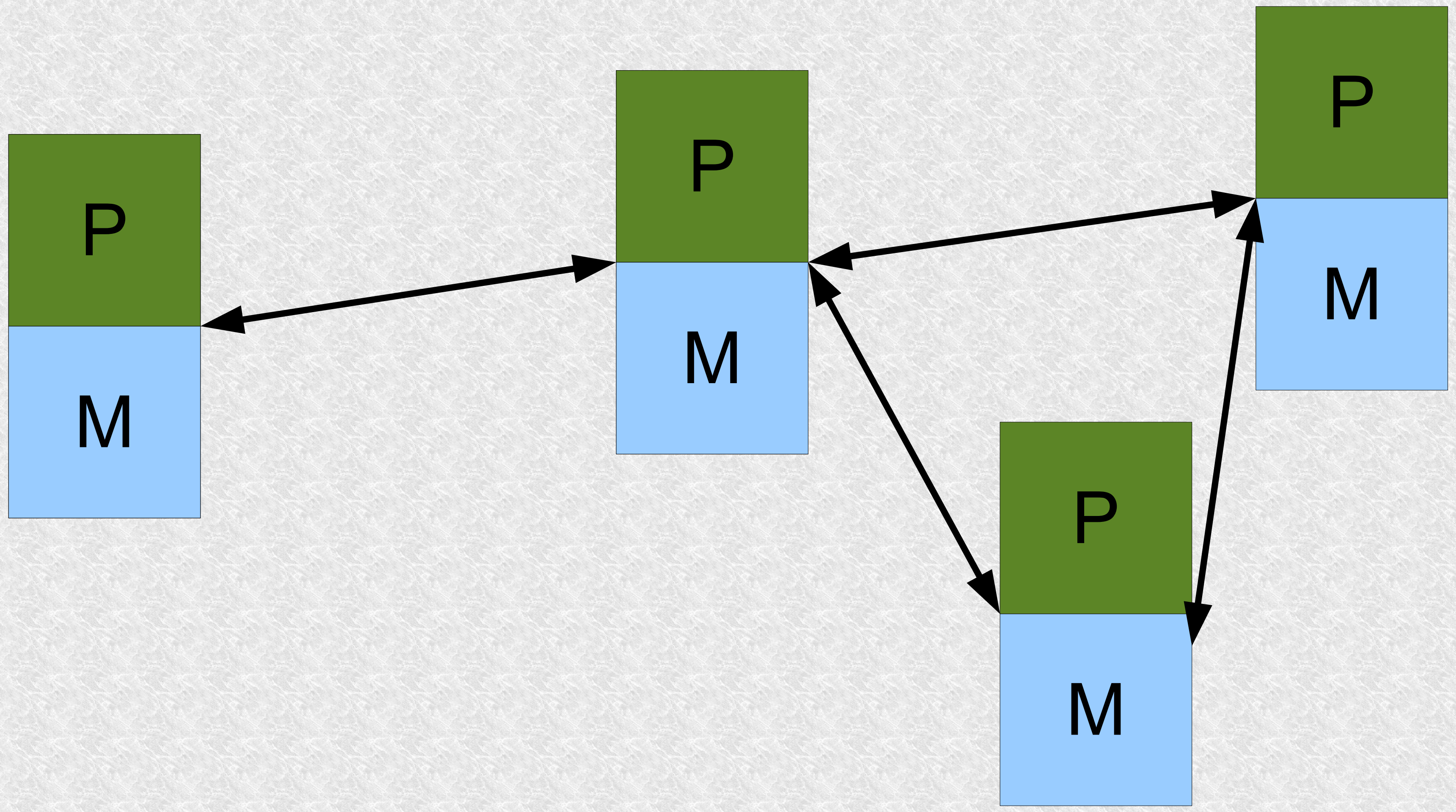
Classification of parallel machines

MIMD – Multiple Instruction Stream, Multiple Data Stream

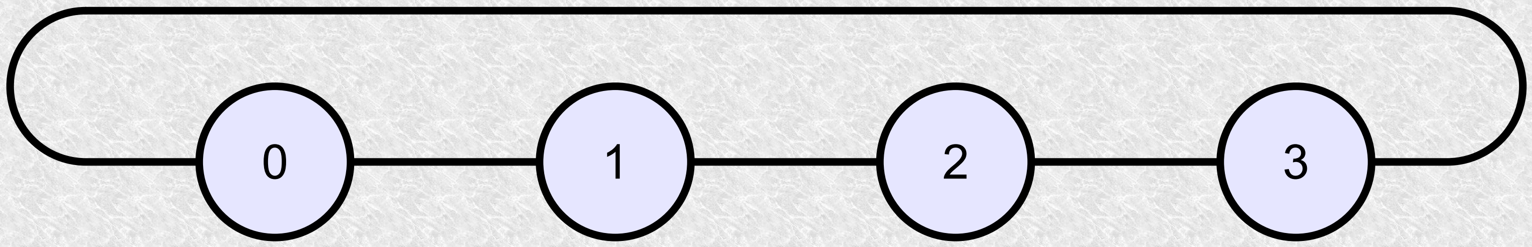


Classification of parallel machines

MIMD – Without shared memory,
a distributed system

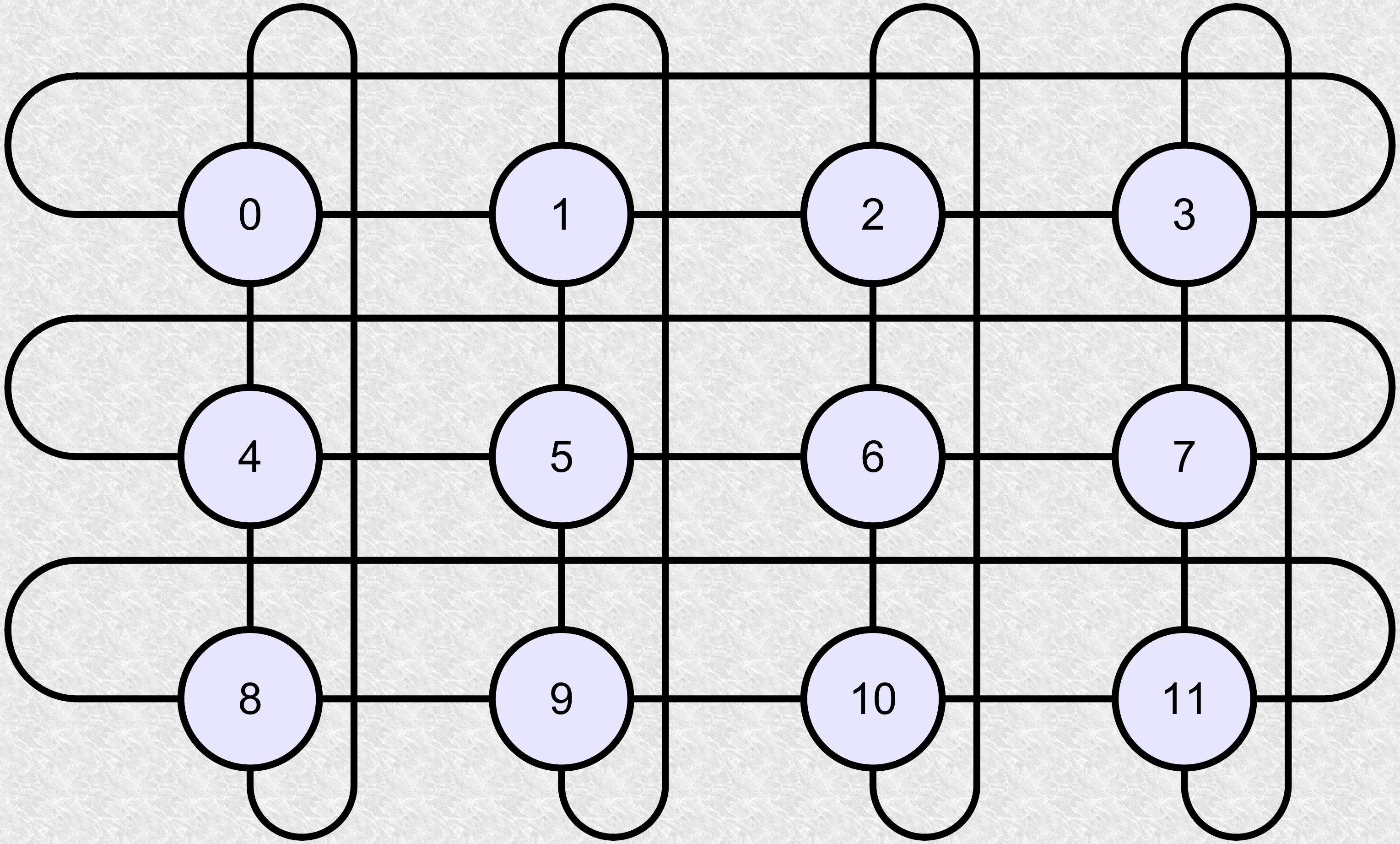


Architecture - the ring



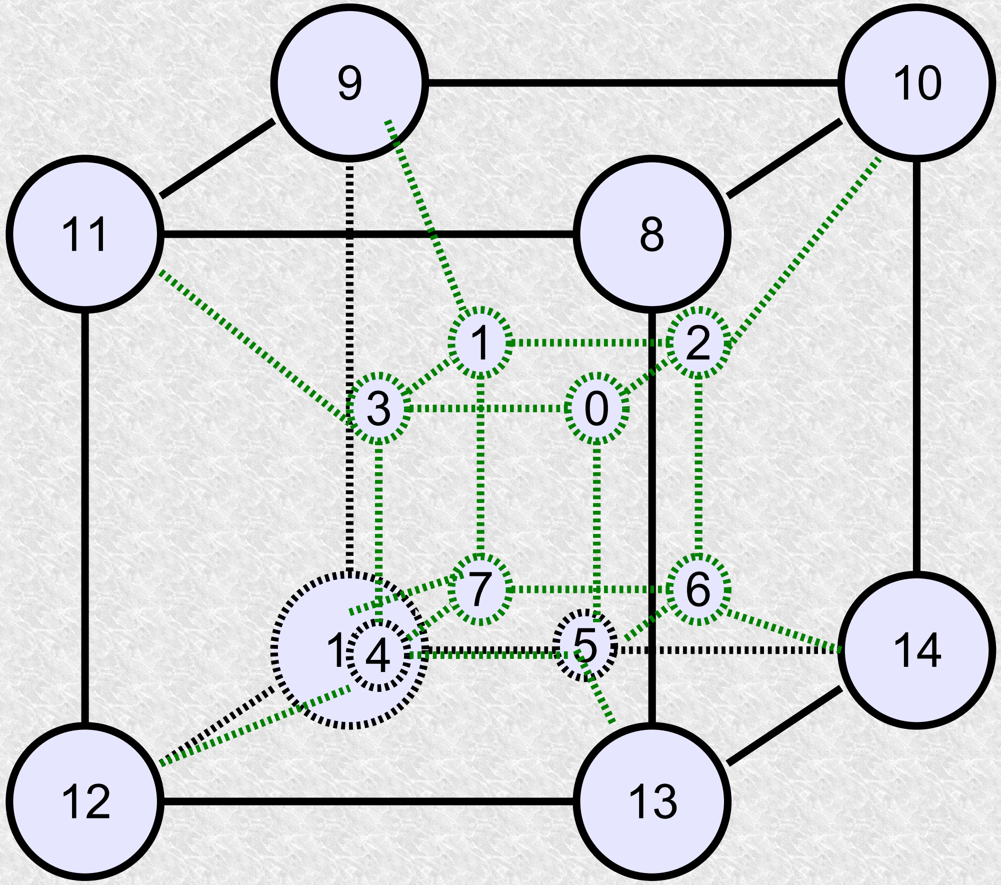
Distance between the two most distant units = 0.5 * p (rounded down) in 2-way transmission and p-1 in unidirectional

Architecture - 2-dimensional table



Distance $0,5*(p_x+p_y)$

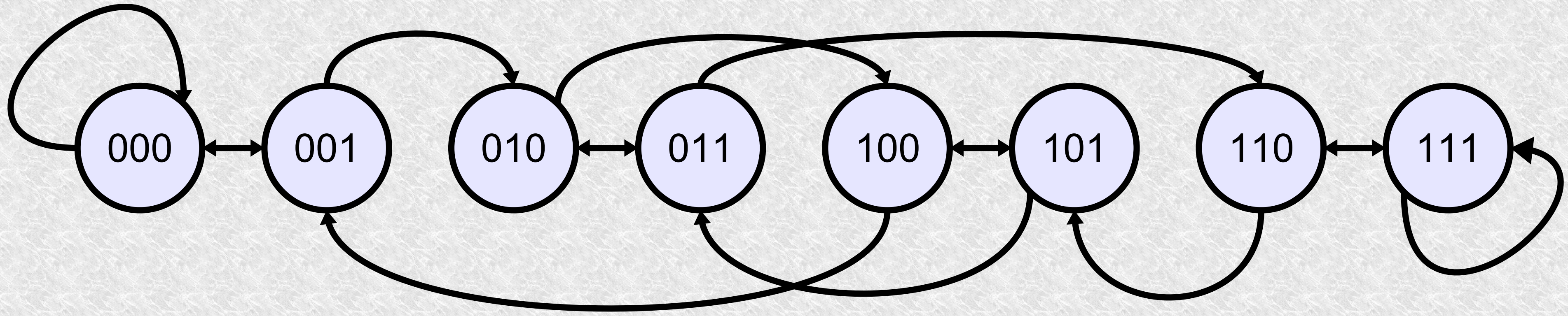
Architecture - Hypercube



Architecture - Hypercube

- A hypercube of order **n** consists of **2n** nodes
- Every higher order architecture includes a lower order architecture
- In the 4-dimensional example shown, we have a $\log_2 p$ distance

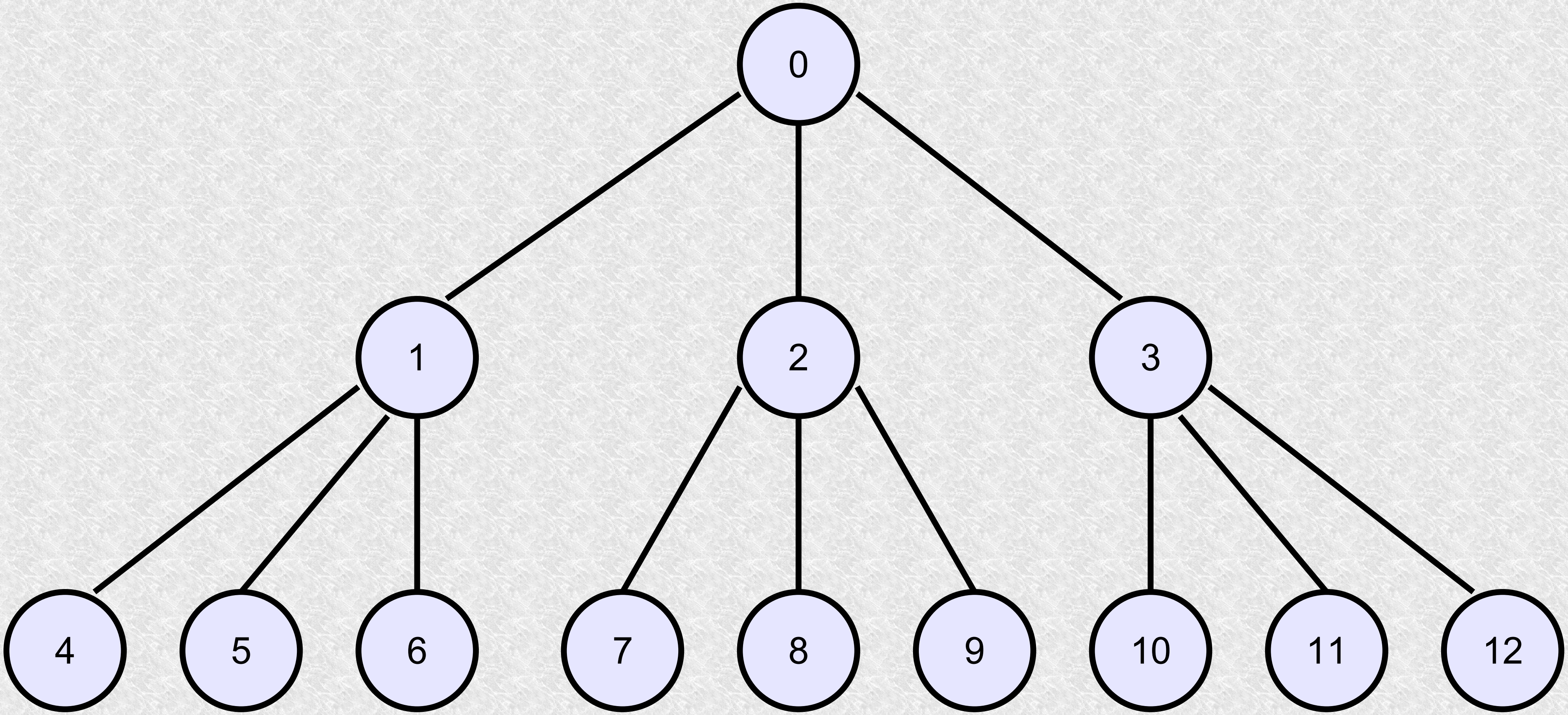
The network perfectly shuffled



The disadvantage of a cube is the logarithmic increase in vertex degrees with its scaling. The perfectly shuffled network consist of z ***p=2ⁿ*** processors.

Two types of links: two-way "exchange" and one-way "shuffle" (from processor ***i*** to ***2i mod (p-1)*** except ***p-1***)

Architecture - tree



Advantage: fewest communication channels = p-1
maximum distance = 2 * levels
The disadvantage: when one node fails, the entire branch stops working

Parallelism classes

Regard to granularity

$$G = \frac{T_{comp}}{T_{comm}}$$

Parallelism classes

- **Fine-grained** parallelism (small G) large number of small tasks. It facilitates load balancing.
- **Coarse-grained** parallelism (big G) large tasks. This might result in load imbalance, wherein certain tasks process the bulk of the data while others might be idle. The advantage of this type of parallelism is low communication and synchronization overhead
- **Medium-grained** parallelism - is a compromise between fine-grained and coarse-grained parallelism

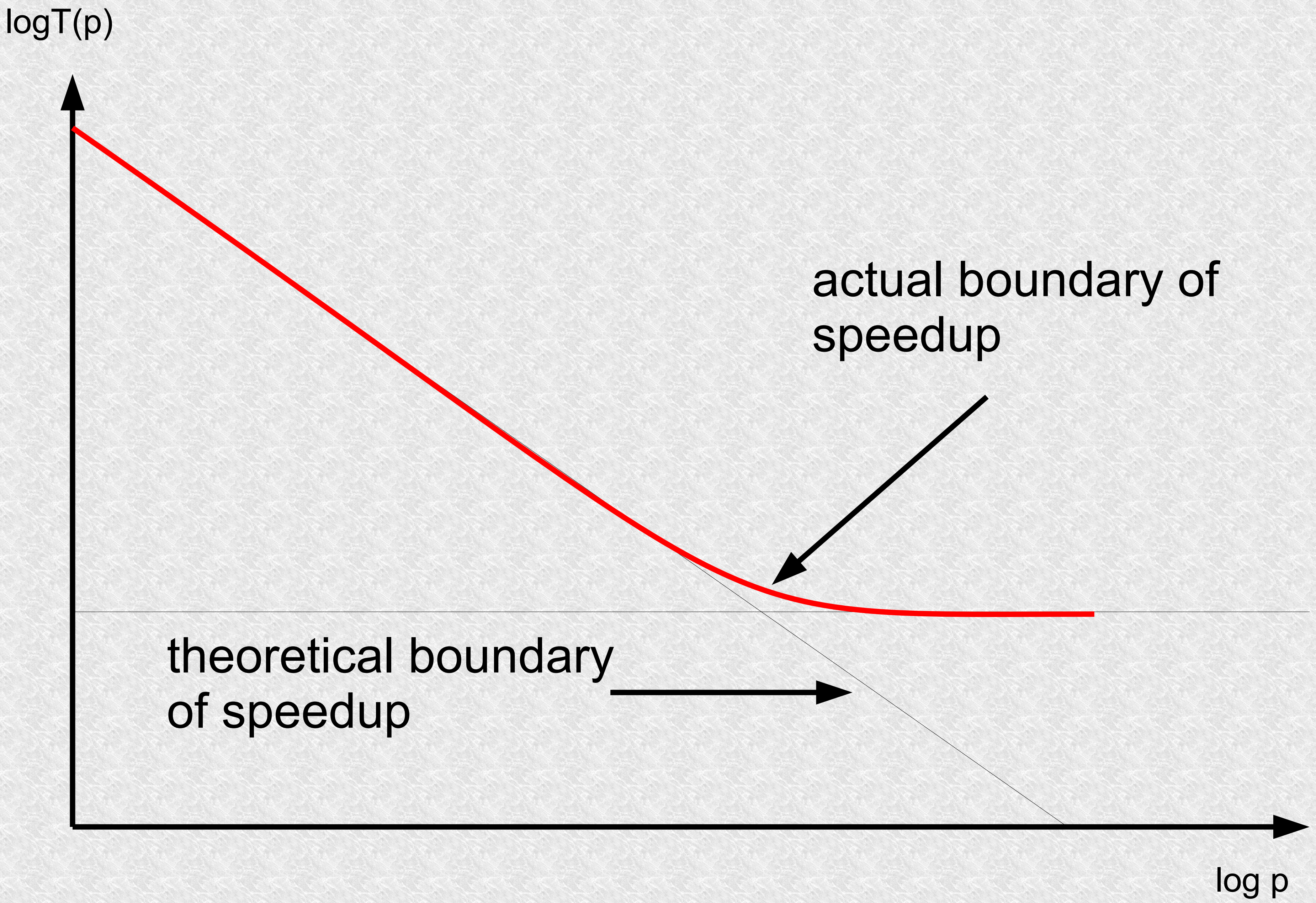
Speedup and Efficiency

Speedup

- To(1) – optimal time of a single-processor solution
- T(p) – time to complete the task by p processors

$$S(p)=\frac{\mathrm{To}(1)}{\mathrm{T}(\mathrm{p})}$$

Limit of Speedup



Efficiency

Efficiency

$$E(p) = \frac{\mathrm{S(p)}}{p} = \frac{\mathrm{To(1)}}{\mathrm{T(p)*p}}$$

In perfect case S(p)=p i E(p)=1

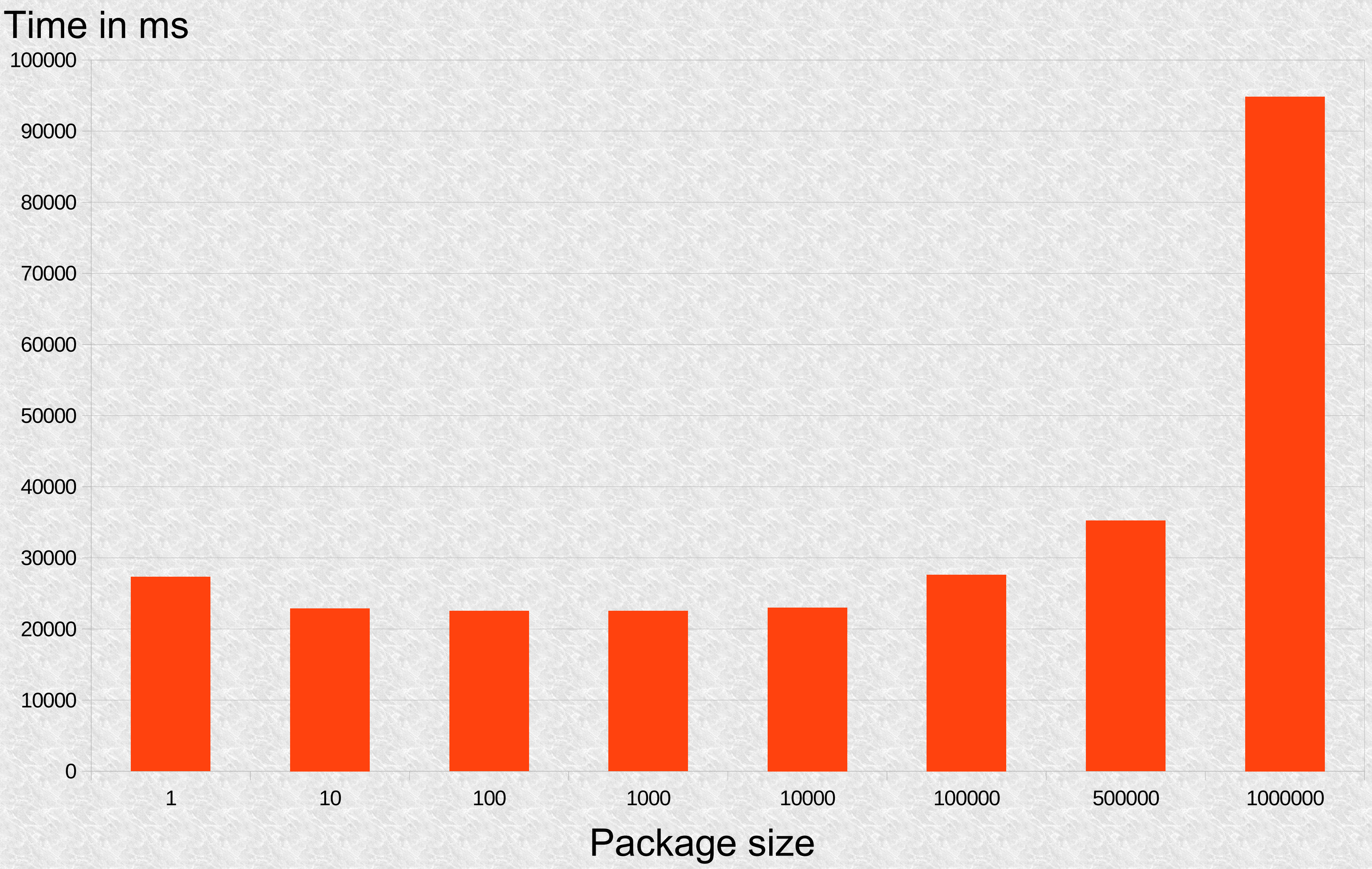
Granularity and Efficiency Example

```
static int Consument()
        {
            int primesCount = 0;
            int min = 0;
            int max = 0;
            int m = 0;

            while (true) //endless
            {
                go.WaitOne(); // wait for data
                min = lowLimit; //remember global var to local
                max = hiLimit;
                ready.Set(); // ok we read data now, server can change lowLimit and hiLimit
                if (min == max) // this is the end
                {
                    Console.WriteLine("finish");
                    go.Set(); // let another consument in
                    return primesCount;
                }

                for (int n = min; n < max; n++)
                {
                    bool isPrime = true;
                    m = n / 2;
                    for (int i = 2; i <= m; i++)
                    {
                        if (n % i == 0)
                        {
                            isPrime = false;
                            break;
                        }
                    }
                    if (isPrime == true)
                        primesCount++;
                }
           //      Console.WriteLine($"Found {primesCount} primes between {min} and {max}");
            }
        }
```

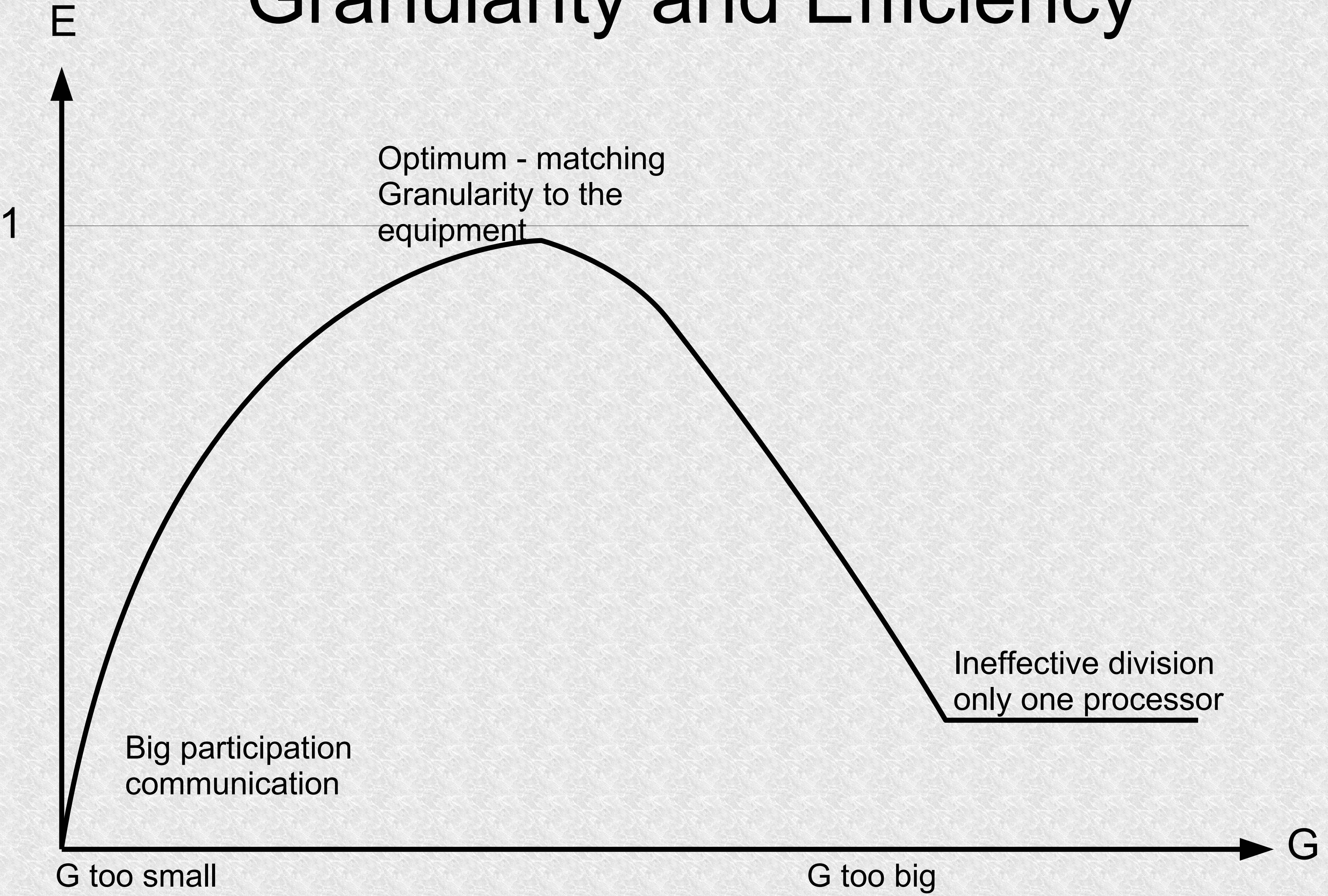
Granularity and Efficiency Example

```
        static int lowLimit = 0;
        static int hiLimit = 0;
        static EventWaitHandle ready = new AutoResetEvent(true);
        static EventWaitHandle go = new AutoResetEvent(false);
        static void Server(int packetSize, int packetCount)
        {

            for (int i = 0; i < packetCount; i++)
            {
                ready.WaitOne(); // wait for consument ready to 
read next data
                lowLimit = hiLimit;
                hiLimit += packetSize;
                go.Set(); // tell consument that data are ready
            }
            ready.WaitOne(); // ensure last values was read
            lowLimit = hiLimit; //it's a sign to end
            go.Set();
            Console.WriteLine("Server finished");
        }
```

Granularity and Efficiency Example



Granularity and Efficiency



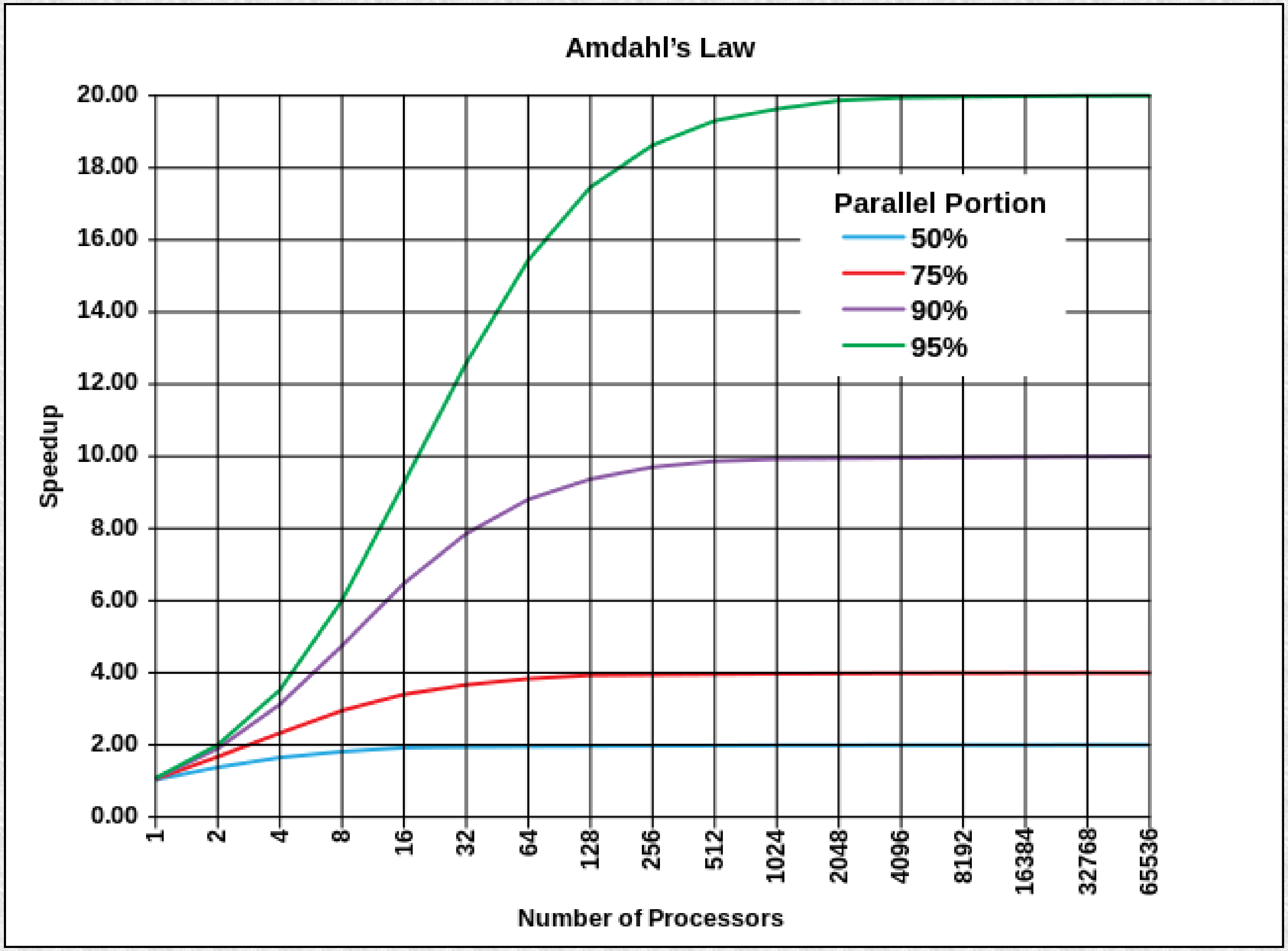
Amdahl's Law

- **P** is a part that does not benefit from the improvement of the resources of the system, runs in sequential way. For example critical section
- **(1-P)** is a part that benefits from the improvement of the resources of the system
- N – processor's count

Maximal speedup:

$$S = \frac{1}{(1-P) + \frac{P}{N}}$$

Amdahl's Law



Source Wikipedia

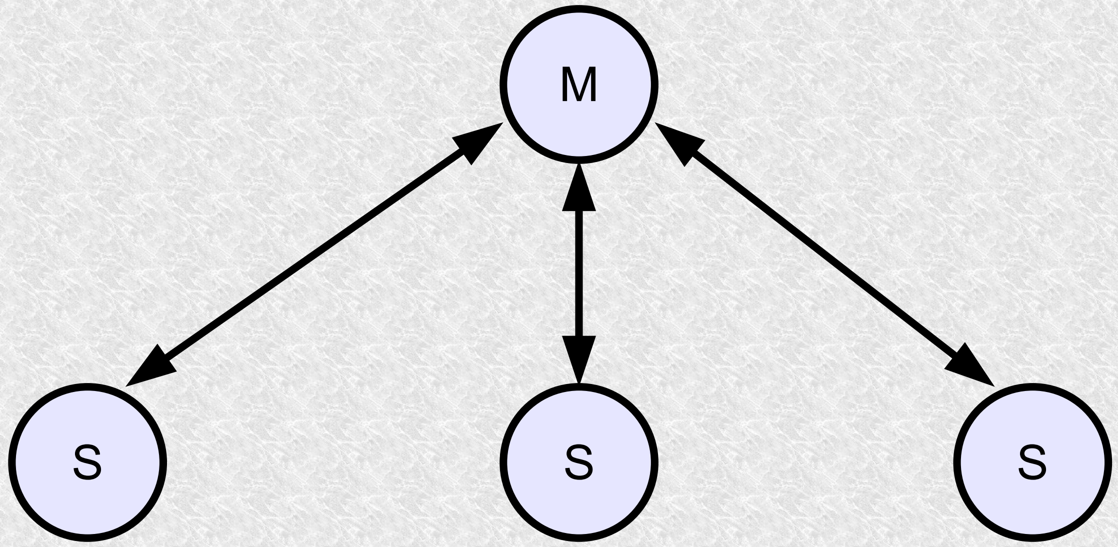
Amdahl's Law

- When we know speedup **S** for **N** processors
- Theoretical **P** we can estimate:

$$P_{est} = \frac{\frac{1}{S} - 1}{\frac{1}{N} - 1}$$

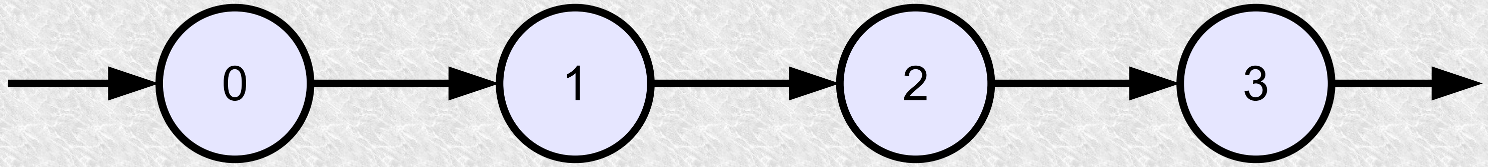
Organization of calculations

System with master-slave process



Organization of calculations

Pipelining



- Each processor must wait for it to receive the data
- cannot start simultaneously
- the entire system runs at its slowest pace
- little flexibility
- N - packet count
- p - processors

$$E = \frac{N*p*T}{p*(N+p-1)*T} = \frac{N}{N+p-1}$$

Why is it worth paralleling

Simple Replace Sort

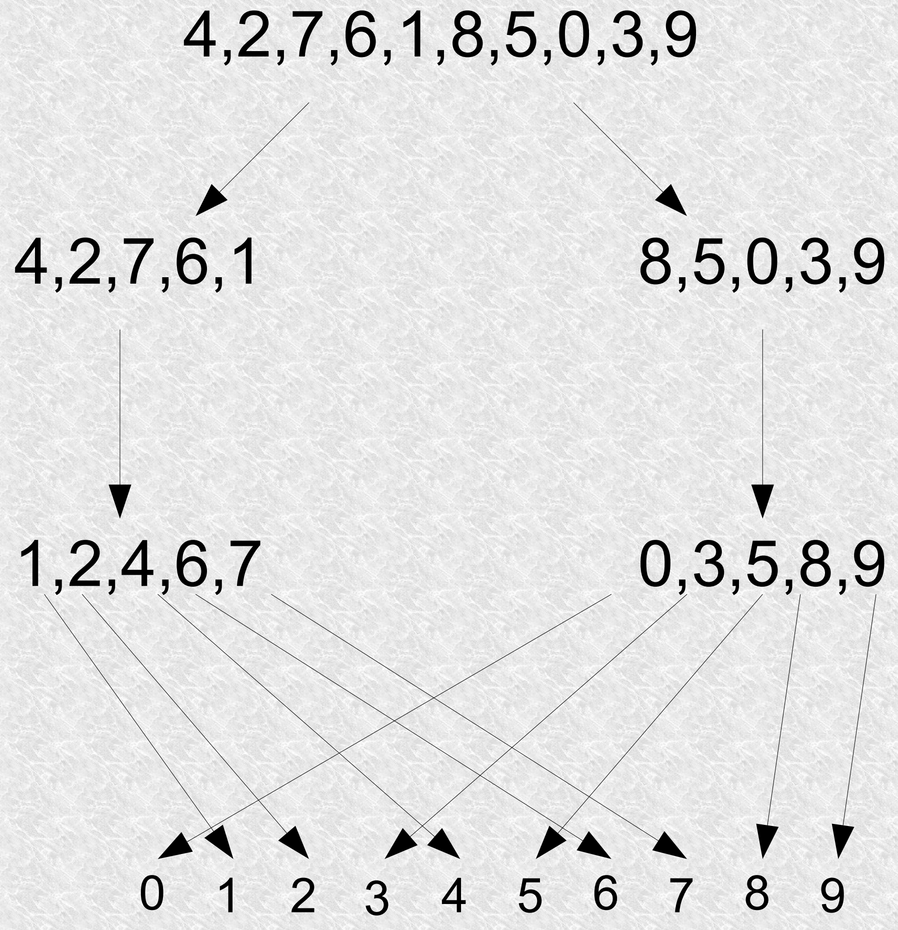
```
public static void Sort(SortData sData)
{

    int temp;
    for (int i = sData.Left; i < sData.Right - 1; i++)
    {
        for (int j = i; j < sData.Right; j++)
        {
            if (sData.ArrayOfInts[j] < sData.ArrayOfInts[i])
            {
                temp = sData.ArrayOfInts[j];
                sData.ArrayOfInts[j] = sData.ArrayOfInts[i];
                sData.ArrayOfInts[i] = temp;
            }
        }
    }
}
```

Simple Replace Sort

Complexity: $(n-1)+(n-2)+...+1=\frac{\text{n(n-1)}}{2}$

It is about: $\frac{n^2}{2}$

Merge sort



Merge sort alghoritm

```
public static int[] Merge(SortData sData)
         {
            int[] arrayOut = new int[sData.Size];
            int counterOut = 0;
            int ind1, ind2;
            int count1 = sData.Left;
            int count2 = sData.Middle;
            while (count1 < sData.Middle)
             {
                if (sData.ArrayOfInts[count1] < sData.ArrayOfInts[count2])
                  {
                      arrayOut[counterOut++] = sData.ArrayOfInts[count1++];
                    if (count1 >= sData.Middle)
                        //left part is finished so rewrite rest of
                        for (ind2 = count2; ind2 < sData.Right; ind2++)
                          {
                              arrayOut[counterOut++] = sData.ArrayOfInts[ind2];
                          }
                  }
                else
                  {
                      arrayOut[counterOut++] = sData.ArrayOfInts[count2++];
                    if (count2 >= sData.Right) //rest of numbers can be rewrite direct
                      {
                        for (ind1 = count1; ind1 < sData.Middle; ind1++)
                          {
                              arrayOut[counterOut++] = sData.ArrayOfInts[ind1];
                          }
                        count1 = sData.Middle; //finish
                      }
                  }
             }
             sData.ArrayOfInts = arrayOut;
            return arrayOut;
         }
```

Merge sort

```
 SortData sdR = new SortData(array, size, size / 2, 0, size); //sort
parameter for left side of the array
 SortData sdL = new SortData(array, size, 0, 0, size / 2); //sort parameter
for right side of the array
 SortData sdM = new SortData(array, size, 0, size / 2, size); //sort
parameter for merge (whole array with middle point).
SimpleSort.Sort(sdL);
SimpleSort.Sort(sdR);
SimpleSort.Merge(sdM);
```

Merge sort

Estimated cost:

To sort $\frac{n}{2}$ elements we do only $\frac{(\frac{n}{2})^2}{2} = \frac{n^2}{8}$

comparisons

For two subarrays it is $\frac{n^2}{4}$ comparisons + ***n*** for merge.

What is more, subarrays can be sort parallelly

example SimpleSort